Self-Randomized Exponentiation Algorithms

[Published in T. Okamoto Eds., *Topics in Cryptology - CT-RSA 2004* , vol. 2964 of *Lecture Notes in Computer Science*, pp. 236–249, Springer-Verlag, 2004.]

Benoît Chevallier-Mames

Gemplus, Card Security Group
La Vigie, Avenue du Jujubier, ZI Athélia IV, 13705 La Ciotat Cedex, France
benoit.chevallier-mames@gemplus.com
http://www.gemplus.com/smart/

Abstract. Exponentiation is a central process in many public-key cryptosystems such as RSA and DH. This paper introduces the concept of self-randomized exponentiation as an efficient means for preventing DPA-type attacks. Self-randomized exponentiation features several interesting properties:

- it is fully generic in the sense that it is not restricted to a particular exponentiation algorithm;
- it is parameterizable: a parameter allows to choose the best trade-off between security and performance;
- it can be combined with most other counter-measures;
- it is space-efficient as only an additional long-integer register is required;
- it is flexible in the sense that it does not rely on certain group properties;
- it does not require the prior knowledge of the order of the group in which the exponentiation is performed.

All these advantages make our method particularly well suited to secure implementations of the RSA cryptosystem in standard mode, on constrained devices like smart cards.

Keywords: Exponentiation, implementation attacks, fault attacks, side-channel attacks (DPA, SPA), randomization, exponent masking, blinding, RSA, standard mode, smart cards.

1 Introduction

Since the invention of the public key cryptography by Diffie and Hellman [DH76], numerous public-key cryptosystems were proposed. Amongst those that resisted cryptanalysis, the RSA cryptosystem [RSA78] is undoubtedly the most widely used. Its intrinsic security relies on the difficulty of factoring large integers. In spite of decades of intensive research, the factoring problem is still considered as

a very hard problem, making the RSA cryptosystem secure for sensitive applications such as data encryption or digital signatures [PKC02].

Instead of trying to break the RSA at a mathematical level, cryptographers then turned their attention to *concrete* implementations of RSA cryptosystems. This gave rise to fault attacks [BDL01] and side-channel attacks [Koc96,KJJ99]. Implementation attacks profoundly modified the way algorithms should be implemented.

As a general rule of thumb for preventing implementation attacks, algorithms should be randomized. In the case of the RSA cryptosystem, there are basically two approaches for randomizing the computation of $y = x^d \pmod N$. This can be achieved by:

1. randomizing the input data prior to executing the exponentiation algorithm [Koc96]; e.g., as
   (a) $\hat{x} \leftarrow x + r_1 N$ for a k-bit random r_1
   (b) $\hat{d} \leftarrow d + r_2 \phi(N)$ for a k-bit random r_2
   and then y is evaluated as $y = \hat{y} \pmod N$ with $\hat{y} = \hat{x}^{\hat{d}} \pmod{2^k N}$;
2. randomizing the exponentiation algorithm itself (e.g., [Wal02], [MDS99]).

The first approach, initiated by Kocher (see [Koc96, Section 10]), presents the advantage of being independent of the exponentiation algorithm. It also is worth noting that when x is the result of a probabilistic padding (e.g., OAEP [BR95] or PSS [BR96]), there is no need to further randomize x and so the exponentiation can, for example, be carried out as $y = x^{\hat{d}} \pmod N$ with $\hat{d} = d + r_2 \phi(N)$ for a random r_2. Unfortunately, such a randomization of d is restricted to CRT implementations of RSA [QC82] as the value of Euler totient function $\phi(N)$ is usually unknown to the private exponentiation algorithm in standard (i.e., non-CRT) mode.[1]

The best representative of the second approach is the MIST algorithm by Walter [Wal02]. MIST randomly generates a fresh addition chain for exponent d for performing $x^d \pmod N$. To minimize the number of registers, the addition chain is computed on-the-fly via an adaptation of an exponentiation algorithm based on "division chains" [Wal98]. Another example is an improved version of the sliding window method proposed in [IYTT02] . Compared to the first approach, it allows to randomize the exponentiation without the knowledge of $\phi(N)$ but requires a secure division algorithm for computing the division chains or quite complicated management.

This paper presents a novel method to randomize the execution of the exponentiation, in order to prevent *Differential Power Analysis* (DPA) [KJJ99], combining the advantages of the two approaches: As in the first approach, it does not impose a particular exponentiation algorithm; and as in the second approach, it is a randomized algorithm (in particular, it does not require the

[1] When the public exponent e is known and not too large, one can randomize the private exponent as $\hat{d} \leftarrow d + r(ed - 1)$. Unfortunately, in most cases, e is unknown (i.e., not available to the private exponentiation algorithm).

knowledge of $\phi(N)$ nor of e in a private RSA exponentiation). Our method introduces the concept of *self-randomized exponentiation*, meaning that exponent d is used itself as an additional source of randomness in the exponentiation process. Self-randomized exponentiation only assumes that exponent bits are scanned from the most significant position and so applies to most exponentiation algorithms [MvV97, Chapter 14]. It can also be combined with most other counter-measures such as randomizing the exponent prior to the exponentiation. Finally, our method is not restricted to exponentiation in RSA groups and equally applies to other groups such as the group of points of an elliptic curve over a finite field [Kob87,Mil86].

The rest of this paper is organized as follows. The next section briefly reviews exponentiation algorithms and presents the general principle behind self-randomized exponentiation. In Section 3, two different, self-randomized exponentiation algorithms (and variants thereof) are detailed. Section 4 presents equivalent versions but without branching instructions, so that *Simple Power Analysis* (SPA) [KJJ99] is also prevented. It also presents a version resisting against a powerful attacker able to "reverse" the exponentiation algorithm along with other further optimizations. Finally, Section 5 concludes the paper.

2 Self-Randomized Exponentiation

2.1 Classical exponentiation algorithms

There exist two main families of exponentiation algorithms for evaluating the value of $y = x^d \pmod N$, according to the direction the bits of exponent d are scanned. This paper is only concerned with left-to-right algorithms (i.e., scanning d from the most significant position to the least significant position), including the square-and-multiply algorithm and its k-ary variants, the sliding-window algorithms, ... (see [MvV97, Chapter 14]). Left-to-right algorithms require fewer memory and allow the use of precomputed powers, $x^i \pmod N$, for speeding up the computation of y.

2.2 General principle

Let $d = (d_l, \ldots, d_0)_2 = \sum_{i=0}^{l} d_i\, 2^i$ (with $d_i \in \{0, 1\}$) denote the binary representation of exponent d. Defining

$$d_{k \to j} := (d_k, \ldots, d_j)_2 = \sum_{k \geq i \geq j} d_i\, 2^{i-j}\,,$$

left-to-right exponentiation algorithms share the common feature that an accumulator is used throughout the computation for storing the value of $x^{d_{l \to i}} \pmod N$ for decreasing i's until the accumulator contains the value of $y = x^{d_{l \to 0}} = x^d \pmod N$.

For example, the square-and-multiply algorithm exploits the recurrence relation

$$x^{d_{l\to i}} = (x^{d_{l\to i+1}})^2 \cdot x^{d_i}$$

with $x^{d_{l\to l}} = x^{d_l}$. Therefore, writing at iteration i the value of $x^{d_{l\to i}}$ in accumulator R_0, we obtain the algorithm of Fig. 1.

Input: $x, d = (d_l, \ldots, d_0)_2$
Output: $y = x^d \pmod N$

$R_0 \leftarrow 1$; $R_1 \leftarrow x$; $i \leftarrow l$
while $(i \geq 0)$ **do**
  $R_0 \leftarrow R_0 \cdot R_0 \pmod N$
  if $(d_i = 1)$ **then** $R_0 \leftarrow R_0 \cdot R_1 \pmod N$
  $i \leftarrow i - 1$
endwhile
return R_0

Fig. 1. Square-and-multiply algorithm

Building on the earlier works of [CJRR99,CJ01], we use an additive splitting of the form

$$x^d = x^{d-a} \cdot x^a$$

for a random a, as a means to mask exponent d. A straightforward application of this splitting is inefficient as it roughly doubles the running time: both x^{d-a} and x^a need to be computed.

The main idea behind self-randomized exponentiation consists in taking (part of) d as a source of randomness. So, random a in the above splitting is chosen equal to $d_{l\to i}$, for a random i, since the value of $x^{d\to i}$ is available in the accumulator and needs not to be computed. There are various ways to apply this idea. The next sections present several realizations.

3 Basic Algorithms

3.1 First algorithm

Our first algorithm relies on the simple observation that, for any $l \geq i_j \geq 0$, we have

$$\begin{aligned}
x^d &= x^{d_{l\to 0}} \\
&= x^{d_{l\to 0} - d_{l\to i_1}} \cdot x^{d_{l\to i_1}} \\
&= x^{(d_{l\to 0} - d_{l\to i_1}) - d_{l\to i_2}} \cdot x^{d_{l\to i_1}} \cdot x^{d_{l\to i_2}} \\
&= \ldots \\
&= x^{(((d_{l\to 0} - d_{l\to i_1}) - d_{l\to i_2}) - d_{l\to i_3}) \cdots - d_{l\to i_f}} \cdot x^{d_{l\to i_1}} \cdot x^{d_{l\to i_2}} \cdot x^{d_{l\to i_2}} \cdots x^{d_{l\to i_f}} \ .
\end{aligned}$$

If the i_j's are randomly chosen, the exponentiation process becomes probabilistic. A Boolean random variable ρ is used to determine whether or not the current loop index i belongs to the set $\{i_1, \ldots, i_f\}$. If so, exponent d is replaced with $d - d_{l \to i_j}$. This is illustrated in the next figure.

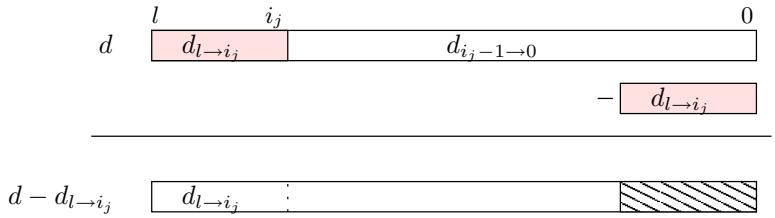

Fig. 2. Masking of exponent d (I)

As in the classical left-to-right exponentiation algorithms, a first accumulator, R_0, is used to keep the value of $x^{d_{l \to i}}$. We also use a second accumulator, R_1, to keep the value of $\prod_{i_j \geq i} x^{d_{l \to i_j}}$. To ensure the correctness of the process, the randomization step $d \leftarrow d - d_{l \to i_j}$ cannot modify the $(l - i_j + 1)$ most significant bits of d (i.e., $d_{l \to i_j}$). This latter condition is guaranteed by checking that $d_{i_j - 1 \to 0} \geq d_{l \to i_j}$ (see Fig. 2).

Applied to the classical square-and-multiply algorithm, we get the following algorithm.

Input: $x, d = (d_l, \ldots, d_0)_2$
Output: $y = x^d \pmod N$

$R_0 \leftarrow 1$; $R_1 \leftarrow 1$; $R_2 \leftarrow x$; $i \leftarrow l$
while $(i \geq 0)$ **do**
  $R_0 \leftarrow R_0 \cdot R_0 \pmod N$
  if $(d_i = 1)$ **then** $R_0 \leftarrow R_0 \cdot R_2 \pmod N$
  $\rho \leftarrow_R \{0, 1\}$
  if $((\rho = 1) \wedge (d_{i-1 \to 0} \geq d_{l \to i}))$ **then**
    $d \leftarrow d - d_{l \to i}$
    $R_1 \leftarrow R_1 \cdot R_0 \pmod N$
  endif
  $i \leftarrow i - 1$
endwhile
$R_0 \leftarrow R_0 \cdot R_1 \pmod N$
return R_0

Fig. 3. Self-randomized square-and-multiply algorithm (I)

Remark 1. In Fig. 3, as at iteration $i = i_j$, the updating step, $d \leftarrow d - d_{l \to i}$, does not modify the $(l - i + 1)$ most significant bits of d, it can be equivalently replaced with $d_{i-1 \to 0} \leftarrow d_{i-1 \to 0} - d_{l \to i}$.

Analysis. We remark that the randomization step (i.e., $d \leftarrow d - d_{l \to i_j}$) modifies the $(l - i_j + 1)$ *least significant* bits of d. Furthermore, the "consistency" condition (i.e., $d_{i_j - 1 \to 0} \geq d_{l \to i_j}$) implies that only about the lower half of exponent d is randomized. For the RSA cryptosystem with small public exponent, this is not an issue since such a system leaks half the most significant bits of the corresponding private exponent d [Bon99, Section 4.5].

A simple variant. The previous methodology applies when the randomization step is generalized to:

$$d \leftarrow d - g \cdot d_{l \to i_j}$$

for some random g such that $d_{i_j - 1 \to 0} \geq g \cdot d_{l \to i_j}$. The second accumulator (say R_1, cf. Fig. 3) should then be updated accordingly as $R_1 \leftarrow R_1 \cdot {R_0}^g \pmod N$. Of particular interest is the value $g = 2^\tau$ as the operation $g \cdot d_{l \to i_j}$ amounts to a shifting and the evaluation of ${R_0}^g \pmod N$ amounts to τ squarings. Again with the example of the square-and-multiply algorithm, we have:

Input: $x, d = (d_l, \ldots, d_0)_2$
Output: $y = x^d \pmod N$

$R_0 \leftarrow 1$; $R_1 \leftarrow 1$; $R_2 \leftarrow x$; $i \leftarrow l$
while $(i \geq 0)$ **do**
  $R_0 \leftarrow R_0 \cdot R_0 \pmod N$
  if $(d_i = 1)$ **then** $R_0 \leftarrow R_0 \cdot R_2 \pmod N$
  $\rho \leftarrow_R \{0, 1\}$; $\tau \leftarrow_R \{0, \ldots, T\}$
  if $((\rho = 1) \wedge (d_{i-1 \to \tau} \geq d_{l \to i}))$ **then**
    $d_{i-1 \to \tau} \leftarrow d_{i-1 \to \tau} - d_{l \to i}$
    $R_3 \leftarrow R_0$
    while $(\tau > 0)$ **do**
      ${R_3} \leftarrow {R_3}^2 \pmod N$; $\tau \leftarrow \tau - 1$
    endwhile
    $R_1 \leftarrow R_1 \cdot R_3 \pmod N$
  endif
  $i \leftarrow i - 1$
endwhile
$R_0 \leftarrow R_0 \cdot R_1 \pmod N$
return R_0

Fig. 4. Self-randomized square-and-multiply algorithm (I')

Note that, at iteration $i = i_j$, the "consistency" condition $d_{i-1 \to 0} \geq 2^\tau \, d_{l \to i}$ is replaced with the more efficient test $d_{i-1 \to \tau} \geq d_{l \to i}$ and the updating step

$d \leftarrow d - 2^{\tau}\, d_{l \to i}$ is replaced with $d_{l \to \tau} \leftarrow d_{l \to \tau} - d_{l \to i} \Leftrightarrow d_{i-1 \to \tau} \leftarrow d_{i-1 \to \tau} - d_{l \to i}$, as mentioned in Remark 1.

Bound T should be chosen as the most appropriate trade-off between the randomization of the most significant bits of d and the efficiency in the evaluation of τ squarings, for a τ randomly drawn in $\{0, \ldots, T\}$.

While it also randomizes the upper half of exponent d, the algorithm of Fig. 4 requires an additional register for computing ${R_0}^{2^{\tau}}$. The next section shows how to remove this drawback.

3.2 Second algorithm

Our first algorithm (Fig. 3) only randomizes the lower half of exponent d as $d \leftarrow d - d_{l \to i_j}$; the restriction coming from the "consistency" condition imposing a half-sized masking. In order to mask the whole value of d, we use the additional trick that

$$d_{l \to i_j - c_j} = (d_{l \to i_j - c_j} - d_{l \to i_j}) + d_{l \to i_j}$$

for any $i_j \geq c_j \geq 0$. Actually, we successively apply the methodology of our first algorithm to sub-exponent $d_{l \to i_j - c_j}$.[2] Moreover, to avoid the use of additional registers, we only perform one randomization at a time. In other words, if we update exponent d as depicted in the next figure

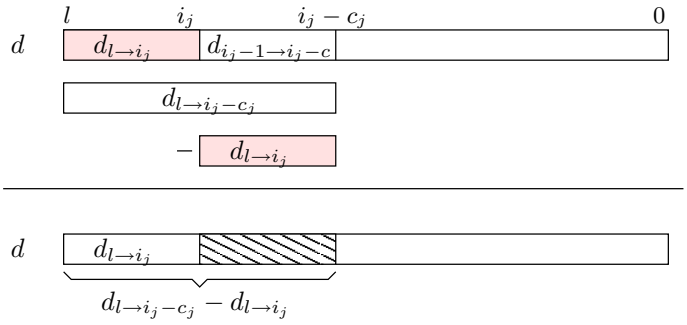

Fig. 5. Masking of exponent d (II)

a new updating step of exponent d will only be permitted after the complete evaluation of $x^{d_{l \to i_j - c_j}} \pmod N$. A Boolean "semaphore", σ, keeps track whether updating is permitted or not.

From Fig. 5, we observe that the $(l - i_j + 1)$ most significant bits of d (i.e., $d_{l \to i_j}$) remain unchanged by the randomization step if

$$\begin{cases} d_{i_j - 1 \to i_j - c_j} \geq d_{l \to i_j}\,, \\ (i_j - 1) - (i_j - c_j) \geq l - i_j \iff c_j \geq l - i_j + 1\ . \end{cases}$$

[2] Our first algorithm corresponds to the case $c_j = i_j, \forall j$.

We set $c_j = l - i_j + 1 + \nu_j$ for some nonnegative integer ν_j. Together with condition $i_j \geq c_j \geq 0$, this implies $2i_j \geq l + 1 + \nu_j$.

Remark 2. If ν_j is equal to 0, the "consistency" condition (i.e., $d_{i_j-1 \to i_j-c_j} \geq d_{l \to i_j}$) is satisfied half of time, approximating $d_{i_j-1 \to i_j-c}$ and $d_{l \to i_j}$ as $(l-i_j+1)$-bit randoms. In other words, if $\nu_j = 0$, half of time randomization is possible. A larger value for ν_j increases the success probability of the consistency condition (and thus of the randomization). On the other hand, it also reduces the possible counter indexes i satisfying the condition $2i_j \geq l + 1 + \nu_j$.

Figure 6 presents the resulting algorithm corresponding to the square-and-multiply algorithm. For all j, the value of ν_j is taken equal to 0 (and thus $c_j = l - i_j + 1$).

Input: $x, d = (d_l, \ldots, d_0)_2$
Output: $y = x^d \pmod N$

$R_0 \leftarrow 1$; $R_1 \leftarrow 1$; $R_2 \leftarrow x$; $i \leftarrow l$; $c \leftarrow -1$; $\sigma \leftarrow 1$
while $(i \geq 0)$ **do**
  $R_0 \leftarrow R_0 \cdot R_0 \pmod N$
  if $(d_i = 1)$ **then** $R_0 \leftarrow R_0 \cdot R_2 \pmod N$
  if $((2i \geq l + 1) \wedge (\sigma = 1))$ **then** $c \leftarrow l - i + 1$ [‡]
  else $\sigma \leftarrow 0$
  $\rho \leftarrow_R \{0, 1\}$
  $\epsilon \leftarrow \rho \wedge (d_{i-1 \to i-c} \geq d_{l \to i}) \wedge \sigma$
  if $(\epsilon = 1)$ **then**
    $R_1 \leftarrow R_0$; $\sigma \leftarrow 0$
    $d_{i-1 \to i-c} \leftarrow d_{i-1 \to i-c} - d_{l \to i}$
  endif
  if $(c = 0)$ **then**
    $R_0 \leftarrow R_0 \cdot R_1 \pmod N$; $\sigma \leftarrow 1$
  endif
  $c \leftarrow c - 1$; $i \leftarrow i - 1$
endwhile
return R_0

Fig. 6. Self-randomized square-and-multiply algorithm (II)

4 Enhanced Algorithms

4.1 Side-Channel atomicity

As presented in the previous section, our algorithms involve numerous branchings and so, although randomized, might be vulnerable to SPA-type attacks [KJJ99].

A generic yet efficient technique, called *"side-channel atomicity"* [CCJ], allows to remove branching conditions at negligible cost. As this is not the main subject of this paper and due to lack of space, we present hereafter, without any further explanation, an atomic version of our first algorithm (Fig. 3). An atomic version of our second algorithm (Fig. 6) can be found in Appendix A.

Input: $x, d = (d_l, \ldots, d_0)_2$
Output: $y = x^d \pmod N$

$R_0 \leftarrow 1$; $R_1 \leftarrow 1$; $R_2 \leftarrow x$; $i \leftarrow l$; $k \leftarrow 0$; $\epsilon = 0$
while $(i \geq 0)$ **do**
  $R_\epsilon \leftarrow R_0 \cdot R_{\epsilon+2k} \pmod N$
  $k \leftarrow k \oplus (d_i \wedge \neg\epsilon)$
  $d \leftarrow d + d_{l\to i} - d_{l\to i} \times (1 + \epsilon)$
  $i \leftarrow i - (\neg k \wedge \neg\epsilon)$
  $\rho \leftarrow_R \{0, 1\}$
  $\epsilon \leftarrow \rho \wedge \neg k \wedge \neg\epsilon \wedge (d_{i-1\to 0} \geq d_{l\to i})$
endwhile
$R_0 \leftarrow R_0 \cdot R_1 \pmod N$
return R_0

Fig. 7. Atomic self-randomized square-and-multiply algorithm (I)

4.2 Reversibility

Throughout this section, we assume that our algorithms are given in a form free of conditional branchings (e.g., by using side-channel atomicity). We will now study their respective strengths against a very powerful imaginary adversary able to distinguish the performed (modular) multiplications. Algorithms I and II involve four types of multiplication:

$$\begin{aligned}
\mathcal{S}: &\quad R_0 \leftarrow R_0 \cdot R_0 \pmod N \\
\mathcal{M}: &\quad R_0 \leftarrow R_0 \cdot R_2 \pmod N \\
\mathcal{C}_1: &\quad R_1 \leftarrow R_0 \cdot R_1 \pmod N \\
\mathcal{C}_2: &\quad R_0 \leftarrow R_0 \cdot R_1 \pmod N
\end{aligned}$$

according to the registers used for the multiplication. Provided that such an attacker makes no errors, Algorithms I and II can be reversed and the value of exponent d recovered. The reversing algorithms are presented in Fig. 8.

We insist that the assumption of recovering the exact sequence of multiplications is unrealistic for present-day cryptographic devices as they include various countermeasures to purposely prevent the distinction between $\mathcal{S}$, $\mathcal{M}$ and $\mathcal{C}_i$. Even under such a strong attack scenario, Algorithm II can be slightly modified in order to make the attack impractical.

Input: $L = (L_{l'}, \ldots, L_0)$
Output: d

$u \leftarrow 0$; $d' \leftarrow 0$; $i \leftarrow l'$
while $(i \geq 0)$ **do**
  case
    $(L_i = \mathcal{S}):$ $d' \leftarrow 2d'$
    $(L_i = \mathcal{M}):$ $d' \leftarrow d' + 1$
    $(L_i = \mathcal{C}_1):$ $u \leftarrow u + d'$
  endcase
  $i \leftarrow i - 1$
endwhile
return $(u + d')$

(a) Algorithm I

Input: $L = (L_{l'}, \ldots, L_0)$
Output: d

$d' \leftarrow 0$; $j \leftarrow l$; $i \leftarrow l'$
while $(i \geq 0)$ **do**
  case
    $(L_i = \mathcal{S}):$ $d' \leftarrow 2d'$
    $(L_i = \mathcal{M}):$ $d' \leftarrow d' + 1$
    $(L_i = \mathcal{C}_2):$ $d' \leftarrow d' + D[\frac{l+j+1}{2}]$
  endcase
  if $(L_{i-1} = \mathcal{S})$ **then** $D[j] \leftarrow d'$; $j \leftarrow j - 1$
  $i \leftarrow i - 1$
endwhile
return d'

(b) Algorithm II

Fig. 8. Recovering exponent d in self-randomized exponentiation algorithms by distinguishing all the involved multiplications

Algorithm II (Fig. 6) is constructed by choosing parameter $\nu_j = 0$ for all j. In fact, parameter ν_j can be any nonnegative integer such that $2i_j \geq l + 1 + \nu_j$ (cf. Remark 2). Hence, the largest possible value for ν_j is $2i_j - l - 1$ and thus, since $\nu_j \geq 0$, parameter $c_j = l - i_j + 1 + \nu_j$ can take any value in the set $\{l - i_j + 1, \ldots, i_j\}$. We generalize our second algorithm by randomly picking c_j in the set $\{l - i_j + 1, \ldots, i_j\}$; i.e., by replacing Line ‡ in Fig. 6 by

$$\textbf{if } ((2i \geq l + 1) \wedge (\sigma = 1)) \textbf{ then } c \leftarrow_R \{l - i + 1, i\}$$

Doing so we obtain a third algorithm (Algorithm III). Its side-channel atomic version is fully given in Appendix A.

Provided that multiplications can be distinguished, reversing Algorithm III translates into the successful execution of the following algorithm:

Since the attacker does not know the random c_j chosen in the set $\{l - i_j + 1, i_j\}$, she has to to try all possible values. Such a exhaustive rapidly becomes impractical, rendering our third algorithm even secure against very powerful adversaries.

4.3 Further optimizations

The frequency of appearance that Boolean variable $\rho = 1$ can be seen as a *tuning parameter* for choosing the best trade-off between performance and security: more randomization penalize the running time and fewer randomization eases the exhaustive search.

A good way to lower the cost of additional operations consists in slightly modifying the random generator outputting ρ so that when Hamming weight of $d - a$ (a may have several definitions according to Algorithm I, II, or III) is

Input: $L = (L_{l'}, \ldots, L_0)$
Output: d

$d' \leftarrow 0$; $j \leftarrow l$; $j_{old} \leftarrow l$; $i \leftarrow l'$
while $(i \geq 0)$ **do**
case
$(L_i = \mathcal{S})$: $d' \leftarrow 2d'$
$(L_i = \mathcal{M})$: $d' \leftarrow d' + 1$
$(L_i = \mathcal{C}_2)$: **for** $\frac{l+j+1}{2} \leq j_{try} \leq j_{old}$, **try** $d' \leftarrow d' + D[j_{try}]$; $j_{old} \leftarrow j$
endcase
if $(L_{i-1} = \mathcal{S})$ **then** $D[j] \leftarrow d'$; $j \leftarrow j - 1$
$i \leftarrow i - 1$
endwhile
return d'

Fig. 9. Exhaustive search on Algorithm III

weaker than Hamming weight of d , ρ has a higher probability of being a 1 and conversely. By this trick, the self-randomized algorithm will tend to select the case which has the weakest Hamming weight, that is, the fastest branch. We note however that the algorithm cannot always select the fastest branch as otherwise it becomes deterministic and so is more easily reversible.

4.4 Average timing

In the following, we give a table with complexity of different algorithms, in term of multiplications.

Table 1. Average number of modular multiplications to perform an exponentiation of length 1024

$\mathcal{S}$, $\mathcal{M}$, $\mathcal{C}_1$, $\mathcal{C}_2$	Square and Multiply		Our algorithms		
	naive	random exp.[3]	(I)	(II)	(III)
Multiplications	1536	1536 + 96	$1536 + 512 \times \bar{\rho}$	1536 + 10	1536 + 10

The overhead factor of Algorithms II and III (10) corresponds in fact to an upper bound of $\log_2 d$. This is a very small quantity but it provides an interesting entropy: the number of possible randomization for a given exponent is superior to $\binom{10}{512} > 2^{64}$.

[3] By random exponent d, we mean the use of $\hat{d}$ as explained in the introduction, with a random r_2 of size 64 bits.

5 Conclusion

This paper introduced the concept of self-randomized exponentiation as an efficient means for preventing DPA-type attacks. Three different such algorithms (and some SPA-protected variants thereof) were described.

Self-randomized exponentiation presents the following interesting properties:

- it is fully generic in the sense that it is not restricted to a particular exponentiation algorithm;
- it is parameterizable: a parameter allows to choose the best trade-off between security and performance;
- it can be combined with most other counter-measures;
- it is space-efficient as only an additional long-integer register is required;
- it is flexible in the sense that it does not rely on certain group properties;
- it does not require the prior knowledge of the order of the group in which the exponentiation is performed.

Of independent interest, the notion of reversibility in self-randomized exponentiation algorithms was defined and a concrete construction was given.

Acknowledgements

The author would like to thank the anonymous referees for their helpful comments that allow us to improve the readability of this paper. Thanks also go to Marc Joye for his careful attention and continuous support in this research.

References

[BDL01] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14(2):101–119, 2001.

[Bon99] Dan Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999.

[BR95] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In A. De Santis, editor, *Advances in Cryptology – EUROCRYPT '94*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer-Verlag, 1995.

[BR96] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures - How to sign with RSA and Rabin. In U. Maurer, editor, *Advances in Cryptology – EUROCRYPT '96*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416. Springer-Verlag, 1996.

[CJ01] Christophe Clavier and Marc Joye. Universal exponentiation algorithm: A first step towards provable SPA-resistance. In Ç.K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 300–308. Springer-Verlag, 2001.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In M. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer-Verlag, 1999.

[CCJ] Benoît Chevallier-Mames, Mathieu Ciet, and Marc Joye. Low cost solutions for preventing simple side-channel power analysis: Side-channel atomicity. To appear. Preprint available on IACR ePrint.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[IYTT02] Kouichi Itoh, Jun Yajima, Masahiko Takenaka, and Naoya Torii. Dpa countermeasures by improving the window method. In Burton S. Kaliski Jr., Ç.K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems– CHES '02*, volume 2523 of *Lecture Notes in Computer Science*, pages 303–317. Springer-Verlag, 2002.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[Koc96] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996.

[MDS99] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Power analysis attacks of modular exponentiation in smartcards. In Ç.K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems– CHES '99*, volume 1717 of *Lecture Notes in Computer Science*, pages 144–157. Springer-Verlag, 1999.

[Mil86] Victor S. Miller. Use of elliptic curves in cryptography. In H.C. Williams, editor, *Advances in Cryptology – CRYPTO '85*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer-Verlag, 1986.

[MvV97] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC Press, 1997.

[PKC02] PKCS #1 v2.1: RSA cryptography standard. RSA Laboratories, June 14, 2002.

[QC82] Jean-Jacques Quisquater and Chantal Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18:905–907, 1982.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[Wal98] Colin D. Walter. Exponentiation using division chains. *IEEE Transactions on Computers*, 47(7):757–765, 1998.

[Wal02] Colin D. Walter. Mist: An efficient, randomized exponentiation algorithm for resisting power analysis. In B. Preneel, editor, *Topics in Cryptology – CT-RSA 2002*, volume 2271 of *Lecture Notes in Computer Science*, pages 53–66. Springer-Verlag, 2002.

A Side-Channel Atomic Exponentiation Algorithms

Input: $x, d = (d_l, \ldots, d_0)_2$
Output: $y = x^d \pmod N$

$R_0 \leftarrow 1$; $R_1 \leftarrow 1$; $R_2 \leftarrow x$; $i \leftarrow l$; $k \leftarrow 0$; $\epsilon = 0$
$\sigma \leftarrow 1$; $c \leftarrow -1$
while $(i \geq 0)$ **do**
  $\theta \leftarrow (c = 0)$
  $R_0 \leftarrow R_0 \cdot R_{\theta+2k} \pmod N$
  $k \leftarrow k \oplus (d_i \wedge \neg\theta)$; $i \leftarrow i - (\neg k \wedge \neg\theta)$
  $\sigma \leftarrow (\sigma \vee \theta) \wedge (2i \geq l + 1)$
  $c \leftarrow \neg\sigma(c - \neg k) + (l - i + 1) \times \sigma$
  $\rho \leftarrow_R \{0, 1\}$
  $\epsilon \leftarrow \rho \wedge \neg k \wedge \sigma \wedge (d_{i-1\to i-c} \geq d_{l\to i})$
  $\sigma \leftarrow \sigma \wedge \neg\epsilon$
  $d_{i-1\to i-c} \leftarrow d_{i-1\to i-c} + d_{l\to i} - d_{l\to i} \times (1 + \epsilon)$
  $R_1 \leftarrow R_{\neg\epsilon}$
endwhile
return R_0

Fig. 10. Atomic self-randomized square-and-multiply algorithm (II)

Input: $x, d = (d_l, \ldots, d_0)_2$
Output: $y = x^d \pmod N$

$R_0 \leftarrow 1$; $R_1 \leftarrow 1$; $R_2 \leftarrow x$; $i \leftarrow l$; $k \leftarrow 0$; $\epsilon = 0$
$\sigma \leftarrow 1$; $c \leftarrow -1$
while $(i \geq 0)$ **do**
  $\theta \leftarrow (c = 0)$
  $R_0 \leftarrow R_0 \cdot R_{\theta+2k} \pmod N$
  $k \leftarrow k \oplus (d_i \wedge \neg\theta)$; $i \leftarrow i - (\neg k \wedge \neg\theta)$
  $\sigma \leftarrow (\sigma \vee \theta) \wedge (2i \geq l + 1)$
  $\gamma \leftarrow_R \{l - i + 1, i\}$
  $c \leftarrow \neg\sigma(c - \neg k) + \gamma \times \sigma$
  $\rho \leftarrow_R \{0, 1\}$
  $\epsilon \leftarrow \rho \wedge \neg k \wedge \sigma \wedge (d_{i-1\to i-c} \geq d_{l\to i})$
  $\sigma \leftarrow \sigma \wedge \neg\epsilon$
  $d_{i-1\to i-c} \leftarrow d_{i-1\to i-c} + d_{l\to i} - d_{l\to i} \times (1 + \epsilon)$
  $R_1 \leftarrow R_{\neg\epsilon}$
endwhile
return R_0

Fig. 11. Atomic self-randomized square-and-multiply algorithm (III)